加餐｜ Rust真的值得我们花时间学习么？

2021-08-24 陈天

《陈天 · Rust 编程第一课》 课程介绍 >

讲述：陈天
时长 16:39 大小 15.26M

你好，我是陈天。

离课程上线到现在，确实没有想到有这么多的同学想要学习 Rust，首先谢谢你的支持、鼓励和反馈。

这两天处理留言，有好多超出我预期的深度发言和问题，比如说 @pedro @有铭 @f 等等同学，不仅让我实实在在地感受到了你们的热情，也让我更加坚定了要教好这门课的决心。正好在这篇加餐中，我来详细谈谈同学们比较关心的一些问题。

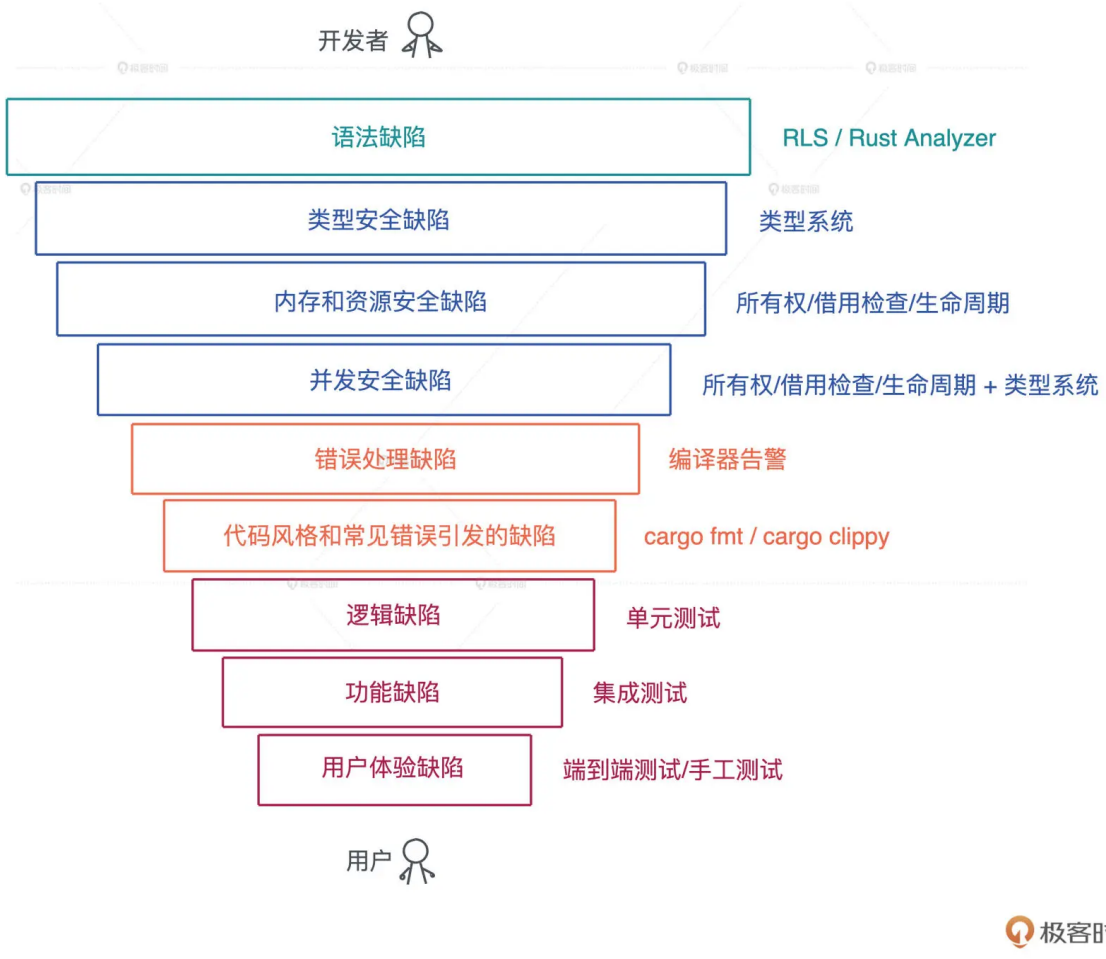
首先会从控制代码缺陷的角度，聊一聊为什么说 Rust 解决了我们开发者在实践过程中遇到的很多问题，而这些问题目前大部分语言都没有很好地解决；然后会再讲讲为什么 Rust 未

来可期，顺便比较一下 Rust 和 Golang，这是留言里问的比较多的；最后还会分享一些 Rust 的学习资料。

代码缺陷

从软件开发的角度来看，一个软件系统想要提供具有良好用户体验的功能，**最基本的要求就是控制缺陷**。为了控制缺陷，在软件工程中，我们定义了各种各样的流程，从代码的格式，到 linting，到 code review，再到单元测试、集成测试、手工测试。

所有这些手段就像一个个漏斗，不断筛查代码，把缺陷一层层过滤掉，让软件在交付到用户时尽善尽美。我画了一张图，将在开发过程中可能出现的缺陷分了类，从上往下看：



语法缺陷

首先在我们开始写代码的时候，在语法层面可能会出现小问题，比如说初学者会对某些语法点不太熟悉，资深工程师在用一些不常用的语法时也会出现语法缺陷。

对于这个缺陷，目前大部分的编程语言都会在你写代码的时候，给到详尽的提示，告诉你语法错误出现在哪里。

对 Rust 来说，它提供了 Rust Language Server / Rust Analyzer 第一时间报告语法错误，如果你用第三方 IDE 如 VSCode，会有这些工具的集成。

类型安全缺陷

然后就是类型方面的缺陷，这类缺陷需要语言本身的类型系统，帮助你把缺陷找出来，所以大部分非类型安全的语言，对这类错误就束手无策了。

以 Python/Elixir 为例，如果你期望函数的参数使用类型 A，但是实际用了类型 B，这种错误只有你的代码在真正运行的时候才能被检查出来，相当于把错误发现的时机大大延后了。

所以现在很多脚本语言也倾向尽可能让开发者多写一些类型标注，但因为它不是语言原生的部分，所以也很难强制，在实际写脚本语言的代码时，你需要特别注意类型安全。

内存和资源安全缺陷

几乎所有的语言中都会有内存安全问题。

对于内存自动管理的语言来说，自动管理机制可以帮你解决大部分内存问题，不会出现内存使用了没有释放、使用了已释放内存、使用了悬停指针等等情况。

我们之前也讲到了，大部分语言，如 Java / Python / Golang / Elixir 等，他们通过语言的运行时解决了内存安全问题。

但是这只是大部分被解决了，还有比如逻辑上存在的内存泄漏的问题，比如一个带 TTL 的缓存，如果没设计好，表中的内容超时后并没有被删除，就会导致内存使用一直增长。这种因为设计缺陷导致的内存泄漏，现在所有语言都没有能够解决这个问题，只能说尽可能地解决。

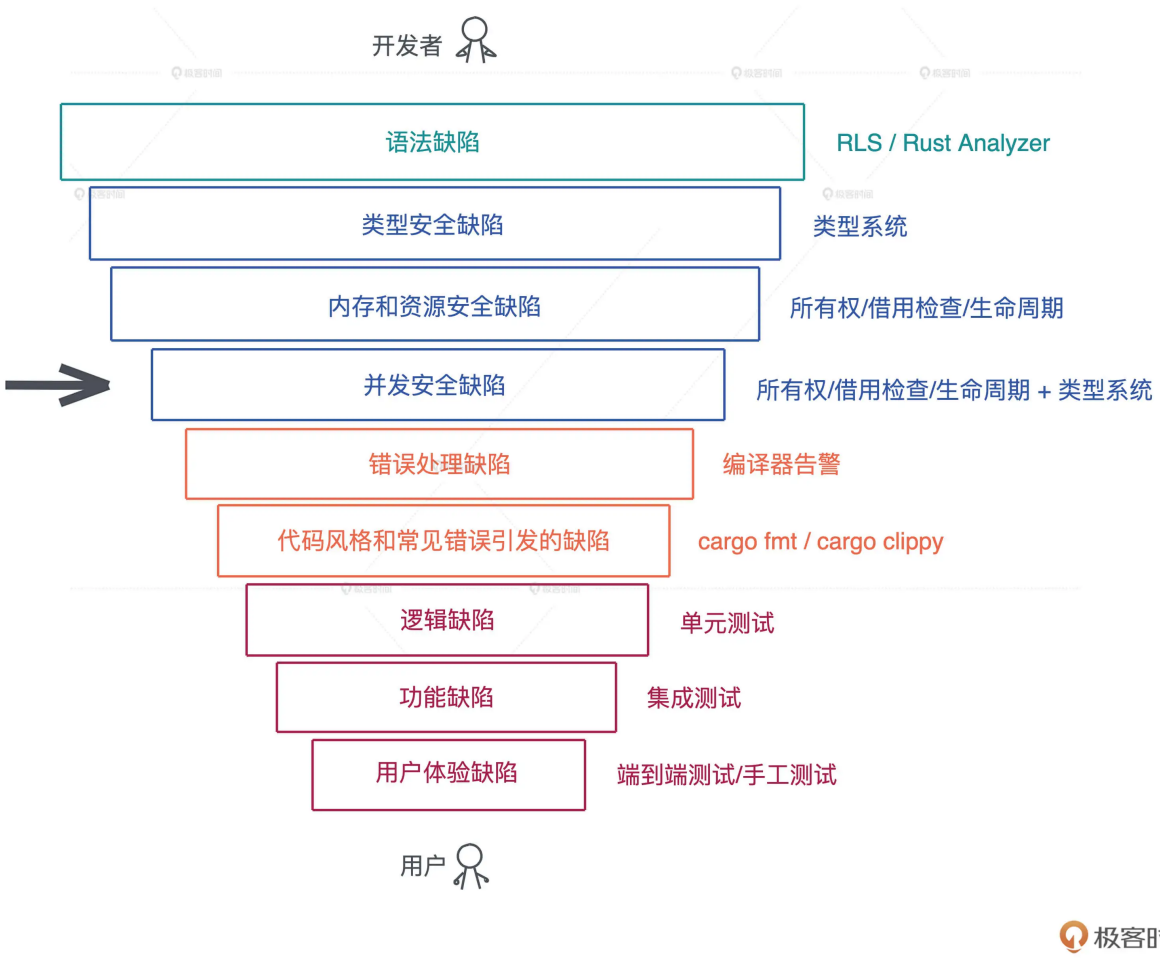
资源安全缺陷也是大部分语言都会有的问题，诸如文件 /socket 这样的资源，如果分配出来但没有很好释放，就会带来资源的泄漏，支持 GC 的语言对此也无能为力，很多时候只

能靠程序员手工释放。

然而资源的释放并不简单，尤其是在做异常处理或者非正常流程的时候，很容易忘记要释放已经分配的资源。

Rust 可以说基本上解决了主要的内存和资源的安全问题，通过所有权、借用检查和生命周期检查，来保证内存和资源一旦被分配，在其生命周期结束时，会被释放掉。

并发安全缺陷



这个问题发生在支持多线程的语言中，比如说两个线程间访问同一个变量，如果没有做合适的临界区保护，就很容易发生并发安全问题。

Rust 通过所有权规则和类型系统，主要是两个 trait：Send/Sync 来解决这个问题。

很多高级语言会把线程概念屏蔽掉，只允许开发者使用语言提供的运行时来保证并发安全，比如 Golang 要使用 channel 和 Goroutine 、Erlang 只能用 Erlang process，只要你在它这个框架下，并发处理就是安全的。

这样可以处理绝大多数并发场景，但遇到某些情况就容易导致效率不高，甚至阻塞其它并发任务。比如当有一个长时间运行的 CPU 密集型任务，使用单独的线程来处理要好得多。

处理并发有很多手段，但是大部分语言为了并发安全，把不少手段都屏蔽了，开发者无法接触到，但是 Rust 都提供给你，同时还提供了很好的并发安全保障，让你可以在合适的场景，安全地使用合适的工具。

错误处理缺陷

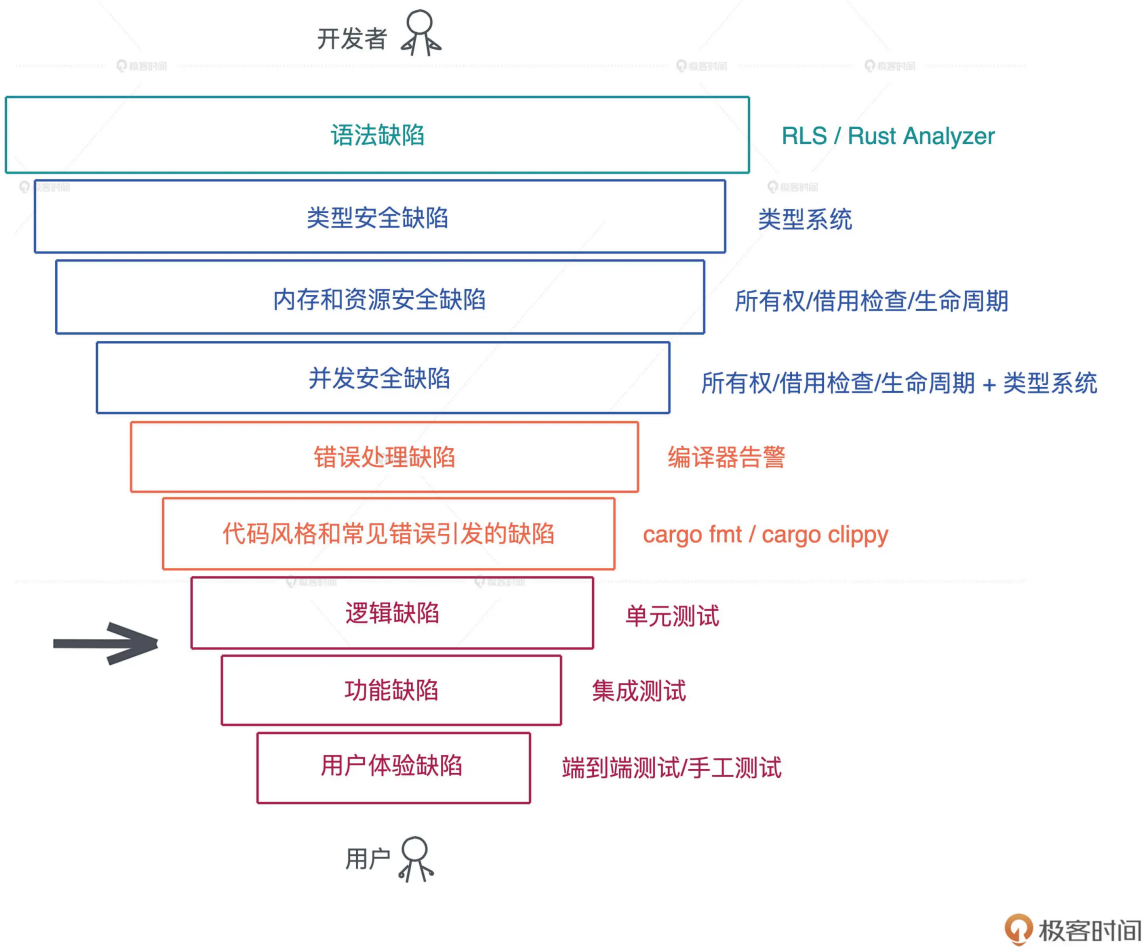
错误处理作为代码的一个分支，会占到代码量的 30% 甚至更多。在实际工程中，函数频繁嵌套的时候，整个过程会变得非常复杂，一旦处理不好就会引入缺陷。常见的问题是系统出错了，但抛出的错误并没有得到处理，导致程序在后续的运行中崩溃。

很多语言并没有强制开发者一定要处理错误，Rust 使用 Result<T, E> 类型来保证错误的类型安全，还强制你必须处理这个类型返回的值，避免开发者丢弃错误。

代码风格和常见错误引发的缺陷

很多语言都会提供代码格式化工具和 linter 来消灭这类缺陷。Rust 有内置的 cargo fmt 和 cargo clippy 来帮助开发者统一代码风格，来避免常见的开发错误。

再往下的三类缺陷是语言和编译器无法帮助解决的。



对于逻辑缺陷，我们需要有不错的单元测试覆盖率；

对于功能缺陷，需要通过足够好的集成测试，把用户主要使用的功能测试一遍；

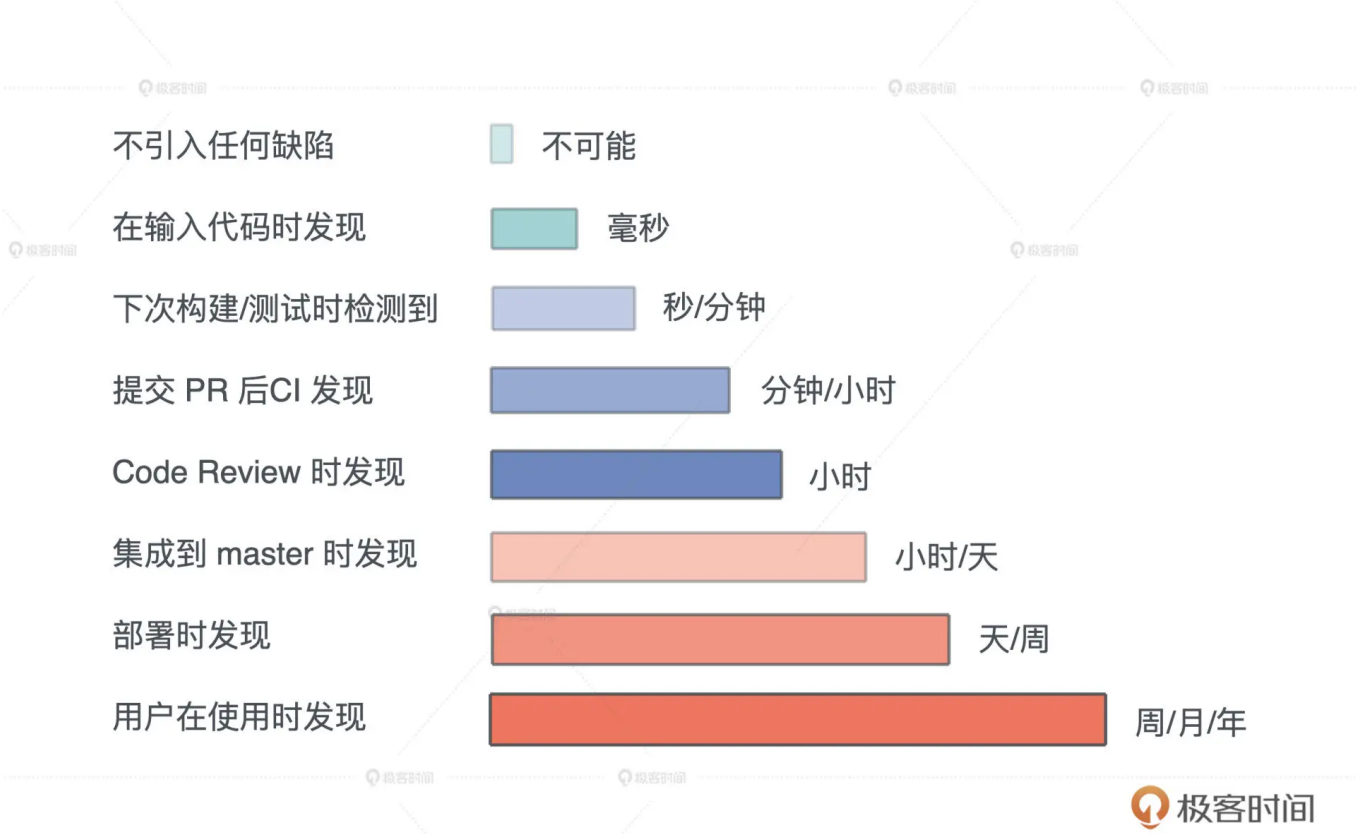
对于用户体验缺陷，需要端到端的测试，甚至手工测试，才能发现。

从上述介绍中你可以看到，Rust 帮我们**把尽可能多的缺陷扼杀在摇篮中**。Rust 在编译时解决掉的很多缺陷，如资源释放安全、并发安全和错误处理方面的缺陷，在其他大多数语言中并没有完整的解决方案。

所以 Rust 这门语言，让开发者的时间和精力都尽可能的放在对逻辑、功能、用户体验缺陷的优化上。

引入缺陷的代价

我们再来从引入缺陷的代价这个角度来看，Rust 这样的处理方式到底有什么好处。



首先，任何系统不引入缺陷是不可能的。

如果在写代码的时候就发现缺陷，纠正的时间是毫秒到秒级；如果在测试的时候检测出来，那可能是秒到分钟级。以此类推，如果缺陷在从 code review 到集成到 master 才被发现，那时间就非常长。

如果一直到用户使用的时候才发现，那可能是以周、月，甚至以年为单位。我之前做防火墙系统时，一个新功能的 bug 往往在一年甚至两年之后，才在用户的生产环境中被暴露出来，这个时候再去解决缺陷的代价就非常大。

所以 Rust 在设计之初，**尽可能把大量缺陷在编译期，在秒和分钟级就替你检测出来，让你修改**，不至于把缺陷带到后续环境，最大程度的保证代码质量。

这也是为什么虽然 Rust 初学者前期需要和编译器做艰难斗争，但这是非常值得的，只要你跨过了这道坎，能够让代码编译通过，基本上你代码的安全性没有太大问题。

语言发展前景判断

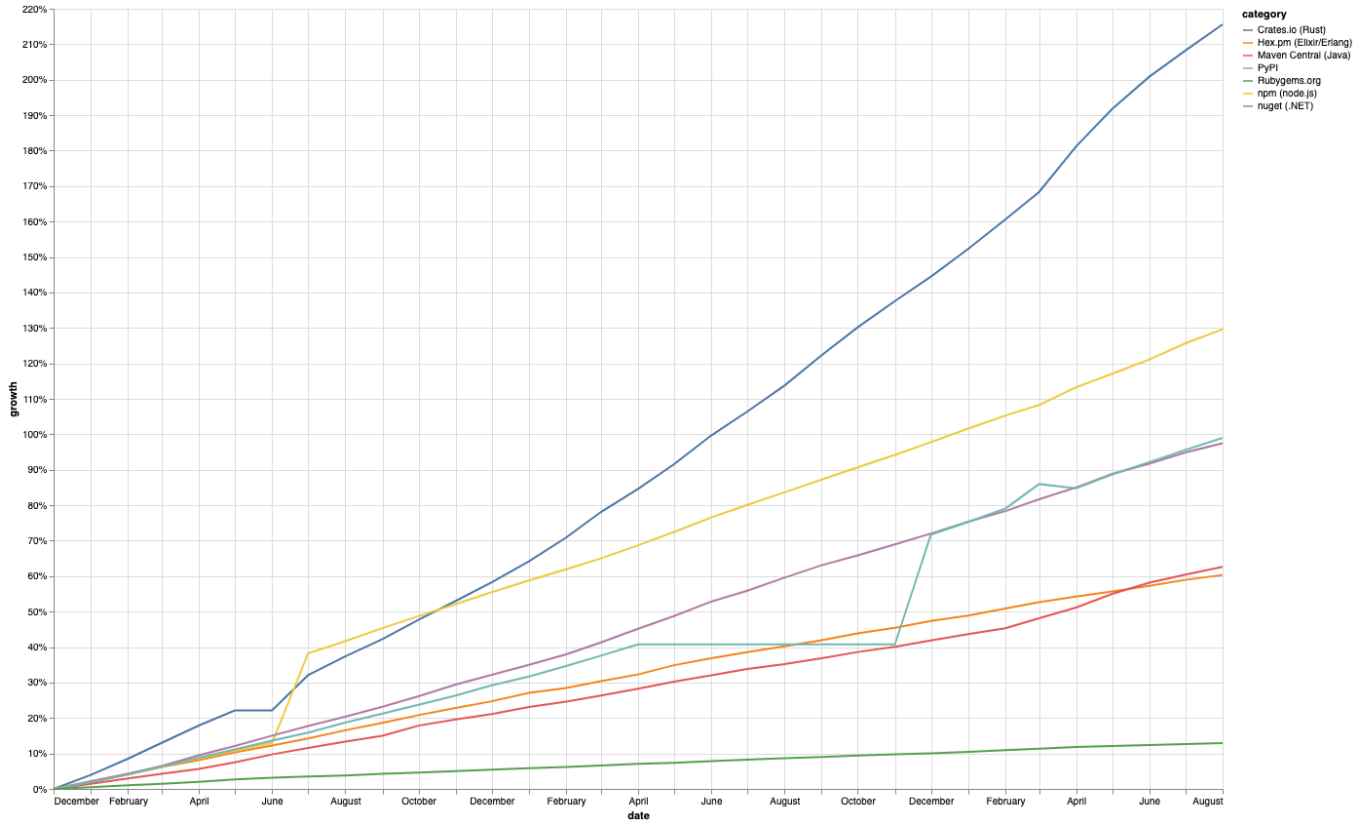
有很多同学比较关心 Rust 的发展前景，留言问 Rust 和其他语言的对比，经常会聊现在或者未来什么语言会被 Rust 替代、Rust 会不会一统前后端天下等等。我觉得不会。

每种语言都有它们各自的优劣和适用场景，谈不上谁一定取代谁。社区的形成、兴盛和衰亡是一个长久的过程，就像“世界上最好的语言 PHP”也还在顽强地生长着。

那么如何判断一门新的语言的发展前景呢？下图是我用 pandas 处理过的 modulecounts 的数据，这个数据统计了主流语言的库的数量。可以看到 2019 年初 Rust crates 的起点并不高，只有两万出头，两年后就有六万多了。

date	Crates.io (Rust)	Hex.pm (Elixir/Erlang)	Maven Central (Java)	npm (node.js)	nuget (.NET)	PyPI	Rubygems.org
2018/12/01	20601.0	7360.0	258811.0	734982.0	135480.0	160265.0	148270.0
2019/01/01	21419.0	7498.0	262534.0	750665.0	138333.0	163292.0	148999.0
2019/02/01	22359.0	7661.0	266448.0	766704.0	141283.0	166760.0	149790.0
2019/03/01	23277.0	7815.0	269822.0	781105.0	144212.0	170256.0	150454.0
2019/04/01	24283.0	7959.0	273407.0	798153.0	148326.0	174331.0	151276.0
2019/05/01	25161.0	8119.0	278170.0	814491.0	151856.0	178139.0	152257.0
2019/06/01	25161.0	8261.0	283985.0	830442.0	155829.0	182054.0	152983.0
2019/07/01	27205.0	8406.0	288759.0	1015666.0	159480.0	185707.0	153514.0
2019/08/01	28304.0	8577.0	293333.0	1041301.0	163054.0	190209.0	153923.0
2019/09/01	29299.0	8733.0	297552.0	1068275.0	166885.0	194286.0	154550.0
2019/10/01	30421.0	8891.0	304997.0	1093050.0	170890.0	198236.0	155098.0
2019/11/01	31521.0	9041.0	309513.0	1117881.0	175334.0	202493.0	155738.0
2019/12/01	32607.0	9179.0	313441.0	1142725.0	179039.0	207107.0	156366.0
2020/01/01	33826.0	9355.0	318565.0	1166944.0	182861.0	211036.0	156929.0
2020/02/01	35202.0	9455.0	322513.0	1190347.0	186915.0	215760.0	157481.0
2020/03/01	36688.0	9595.0	326888.0	1212805.0	191483.0	220474.0	158037.0
2020/04/01	38030.0	9736.0	331899.0	1240034.0	196663.0	225607.0	158732.0
2020/05/01	39465.0	9928.0	337090.0	1267538.0	201507.0	225607.0	159205.0
2020/06/01	41134.0	10074.0	341740.0	1297522.0	207063.0	225607.0	159855.0
2020/07/01	42525.0	10197.0	346377.0	1323647.0	211175.0	225607.0	160477.0
2020/08/01	44035.0	10318.0	349931.0	1349657.0	216203.0	225607.0	161068.0
2020/09/01	45802.0	10445.0	354139.0	1376082.0	220938.0	225607.0	161622.0
2020/10/01	47419.0	10587.0	358717.0	1401507.0	224704.0	225607.0	162265.0
2020/11/01	48939.0	10706.0	362494.0	1427107.0	228906.0	225607.0	162732.0
2020/12/01	50362.0	10847.0	367189.0	1453690.0	233037.0	275178.0	163201.0
2021/01/01	51971.0	10959.0	371748.0	1481737.0	237554.0	280926.0	163715.0
2021/02/01	53683.0	11103.0	376075.0	1508867.0	241627.0	286978.0	164527.0
2021/03/01	55276.0	11235.0	383305.0	1530451.0	246108.0	298037.0	165094.0
2021/04/01	57930.0	11351.0	391200.0	1567491.0	250664.0	296139.0	165757.0
2021/05/01	60101.0	11457.0	401171.0	1595562.0	255836.0	302403.0	166221.0
2021/06/01	61978.0	11576.0	409289.0	1624932.0	259718.0	307966.0	166595.0
2021/07/01	63491.0	11699.0	415277.0	1658603.0	264050.0	313476.0	167011.0
2021/08/01	65019.0	11800.0	420964.0	1687759.0	267522.0	318927.0	167394.0

作为一门新的语言，Rust 生态虽然绝对数量不高，但增长率一直遥遥领先，过去两年多的增长速度差不多是第二名 NPM 的两倍。很遗憾，Golang 的库没有一个比较好的统计渠道，所以这里没法比较 Golang 的数据。但和 JavaScript / Java / Python 等语言的对比足以说明 Rust 的潜力。



Rust 和 Golang

很多同学关心 Rust 和 Golang 的对比，其实网上有很多详尽的分析，这一篇比较不错可以看看。我这里也简单说一下。

Rust 和 Golang 重叠的领域主要在服务开发领域。

Golang 的优点是简单、上手快，语言已经给你安排好了并发模型，直接用即可。对于日程紧迫、有很多服务要写，且不在乎极致性能的开发团队，Golang 是不错的选择。

Golang 因为设计之初要考虑如何能适应新时代的并发需求，所以使用了运行时、使用调度器调度 Goroutine ，在 Golang 中内存是不需要开发者手动释放的，所以运行时中还有 GC 来帮助开发者管理内存。

另外，**为了语法简便，在语言诞生之初便不支持泛型**，这也是目前 Golang 最被诟病的一点，因为一旦系统复杂到一定程度，你的每个类型都需要做一遍实现。

Golang 可能会在 2022 年的 1.18 版本添加对泛型的支持，但泛型对 Golang 来说是一把达摩克利斯之剑，它带来很多好处，但同时会大大破坏 Golang 的简洁和极速的编译体

验，到时候可能会带给开发者这样一种困惑：既然 Golang 已经变得不简单，不那么容易上手，我为何不学 Rust 呢？

Rust 的很多设计思路和 Golang 相反。

Go 相对小巧，类型系统很简单；而 Rust 借鉴了 Haskell，有完整的类型系统，支持泛型。为了性能的考虑，**Rust 在处理泛型函数的时候会做单体化**，泛型函数里每个用到的类型会编译出一份代码，这也是为什么在编译的时候 Rust 编译速度如此缓慢。

Rust 面向系统级的开发，Go 虽然想做新时代的 C，但是它并不适合面向系统级开发，使用场景更多是应用程序、服务等的开发，因为它的庞大的运行时，决定了它不适合做直接和机器打交道的底层开发。

Rust 的诞生目标就是取代 C/C++，想要做出更好的系统层面的开发工具，**所以在语言设计之初就要求不能有运行时**。所以你看到的类似 Golang 运行时的库比如 Tokio，都是第三方库，不在语言核心中，这样可以把是否需要引入运行时的自由度给到开发者。

Rust 社区里有句话说得好：

> Go for the code that has to ship tomorrow, Rust for the code that has to keep running for the next five years.

所以，我对 Rust 的前途持非常乐观的态度。它在系统开发层面可以取代一部分 C/C++ 的场景、在服务开发层面可以和 Java/Golang 竞争、在高性能前端应用通过编译成 WebAssembly，可以部分取代 JavaScript，同时，它又可以方便地通过 FFI 为各种流行的脚本语言提供安全的、高性能的底层库。

我觉得在整个编程语言的生态里，未来 Rust 会像水一样，无处不在且善利万物。

最后给你分享一下我在学习 Rust 的过程中觉得不错的一些资料，也顺带会说明怎么配合这门课程使用。

官方学习资料

Rust 社区里就有大量的学习资料供我们使用。

首先是官方的 Rust book，它涵盖了语言的方方面面，是入门 Rust 最权威的免费资料。不过这本书比较细碎，有些需要重点解释的内容又一笔带过，让人读完还是云里雾里的。

我记得当时学习 Deref trait 时，官方文档这段文字直接把我看懵了：

> Rust does deref coercion when it finds types and trait implementations in three cases:
>
> From &T to &U when T: Deref<Target=U>
>
> From &mut T to &mut U when T: DerefMut<Target=U>
>
> From &mut T to &U when T: Deref<Target=U>

所以我觉得这本书适合学习语言的概貌，对于一时理解不了的内容，需要自己花时间另找资料，或者自己通过练习来掌握。在学习课程的过程中，如果你想巩固所学的内容，可以翻阅这本书。

另外一本官方的 Rust 死灵书（The Rustonomicon），讲述 Rust 的高级特性，主要是如何撰写和使用 unsafe Rust，内容不适合初学者。建议在学习完课程之后，或者起码学完进阶内容之后，再阅读这本书。

Rust 代码的文档系统 docs.rs 是所有编程语言中使用起来最舒服，也是体验最一致的。无论是标准库的文档，还是第三方库的文档，都是用相同的工具生成的，非常便于阅读，你自己撰写的 crate，发布后也会放在 docs.rs 里。在平时学习和撰写代码的时候，用好这些文档会对你的学习效率和开发效率大有裨益。

标准库的文档 建议你在学到某个数据类型或者概念时再去阅读，在每一讲中涉及的内容，我都会放上标准库的链接，你可以延伸阅读。

为了帮助 Rust 初学者进一步巩固 Rust 学习的效果，Rust 官方还出品了 rustlings，它涵盖了大量的小练习，可以用来夯实对知识和概念的理解。有兴趣、有余力的同学可以尝试一下。

其他学习资料

说完了官方的资料，我们看看其它关于 rust 的内容包括书籍、博客、视频。

首先讲几本书。第一本是汉东的《Rust 编程之道》，详尽深入，是不可多得的 Rust 中文书。汉东在极客时间有一门 Rust 视频课程，如果你感兴趣，也可以订阅。英文书有 Programming Rust，目前出了第二版，我读过第一版，写得不错，面面俱到，适合从头读到尾，也适合查漏补缺。

除了书籍相关的资料，我还订阅了一些不错的博客和公众号，也分享给你。博客我主要会看 This week in Rust，你可以订阅其邮件列表，每期扫一下感兴趣的主题再深度阅读。

公众号主要用于获取信息，可以了解社区的一些动态，有 Rust 语言中文社区、Rust 碎碎念，这两个公众号有时会推 This week in Rust 里的内容，甚至会有翻译。

还有一个非常棒的内容来源是 Rust 语言开源杂志，每月一期，囊括了大量优秀的 Rust 文章。不过这个杂志的主要受众，我感觉还是对 Rust 有一定掌握的开发者，建议你在学完了进阶篇后再读里面的文章效果更好。

在 Rust 社区里，也有很多不错的视频资源。社区里不少人推荐 Beginner’ s Series to: Rust，这是微软推出的一系列 Rust 培训，比较新。我简单看了一下还不错，讲得有些慢，可以 1.5 倍速播放节省时间。我自己主要订阅了 Jon Gjengset 的 YouTube 频道，他的视频面向中高级 Rust 用户，适合学习完本课程后再去观看。

国内视频的话，在 bilibili 上，也有大量的 Rust 培训资料，但需要自己先甄别。我做了几期“程序君的 Rust 培训”感兴趣也可以看看，可以作为课程的补充资料。

说这么多，希望你能够坚定对学习 Rust 的信心。相信我，**不管你未来是否使用 Rust，单单是学习 Rust 的过程，就能让你成为一个更好的程序员**。

欢迎你在留言区分享你的想法，我们一起讨论。

参考资料

1. 配合课程使用：官方的 Rust book、微软推出的一系列 Rust 培训 Beginner’ s Series to: Rust、英文书 Programming Rust 查漏补缺

2. 学完课程后进阶学习：官方的 Rust 死灵书（The Rustonomicon）、每月一期的 Rust 语言开源杂志、 Jon Gjengset 的 YouTube 频道、张汉东的《Rust 编程之道》、我的 B 站上的“程序君的 Rust 培训”系列。

3. 学有余力的练习：Rust 代码的文档系统 docs.rs 、小练习 rustlings

4. 社区动态：博客 This week in Rust 、公众号 Rust 语言中文社区、 公众号 Rust 碎碎念

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 13 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 02 | 串讲：编程开发中，那些你需要掌握的基本概念

下一篇 03 | 初窥门径：从你的第一个Rust程序开始！

精选留言 (13)

写留言

leiiiooo 置顶

2021-08-25

老师好，今天阅读到这段话的时候不是很理解。 “Rust 的诞生目标就是取代 C/C++，想要做出更好的系统层面的开发工具，所以在语言设计之初就要求不能有运行时” 这个不能有运行时如何理解，C/C++没有运行时吗？这里我的记忆有些模糊了，也去查了些资料，还是不太理解，Rust也没有运行时吗？没有运行时它是直接由编译器生成到机器码吗？

展开

作者回复: 对，Rust 和 C/C++ 都是编译成机器码，直接面对具体的 CPU 架构。所以 Rust 代码需要为每个平台单独编译，这是它和 Java/DotNet 等语言的主要区别；但是像 golang 这样的语言，它也直接编译成机器码，但 golang，即便一个最简单的 hello world，编译出来的代码也包含了一个庞大的运行时，处理调度，GC 等工作。Rust 没有这些额外的运行时。

2 3

小样 置顶

2021-08-25

再推荐一个和“rustlings”很像的，也是跟着一个个的例子往前走，《通过例子学rust》，有中文版。

另外rust看起来确实很美好，内存安全，无gc，无畏并发。但我不相信这世上有银弹，一门语言就解决了所有问题，继续学习吧，希望陈老师能带我们看看rust这个洞到底有多...

展开

作者回复: 是的，一门语言不能解决所有问题。:)

pedro

2021-08-25

这里补充一点，即使你看了再多的书，刷了再多的视频，你的rust其实还是一地稀碎。

纸上得来终觉浅，绝知此事要躬行，所以没有什么是比和编译期做斗争，再思考，再斗争更好的学习方式了！

展开

作者回复: 对的。所以我们强调要实践，要学以致用。

1 13

MacJayLee

2021-08-25

太值得学习了，为数不多的性能与内存安全并存的语言， 我是重度golang开发者，golang很优秀，网络io型程序开发的绝对利器，开发效率和性能可以达到一个很好的平衡，但是面对内存的过高占用以及cpu密集型计算的场景，深感无力，所以必须掌握rust。相信go和rust的结合一定会有光明的未来

展开

作者回复: 的确如此

7

granagrapn
2021-08-25

学习资料推荐一个: cheats.rs
里面有语法特性概览, 适合速查

作者回复: 对，这个非常棒，我在后续的讲解中还引用了其中的图。还有一个类似的是：https://docs.google.com/presentation/d/1q-c7UAyrUlM-eZyTo1pd8SZ0qwA_wYxmPZVOQkoDmH4/

5

Christopher
2021-08-25

听音频最后有惊喜哦:)

展开

作者回复: :)

1 2

官
2021-08-25

自己学习Rust的动机就是：已经通过Python用了不少“轮子”，在C++/Rust中先学Rust了解语言外知识造“轮子”。现在看来是选对了。

展开

作者回复: 嗯，Rust 特别适合为 Python 造轮子。pyo3 (https://github.com/PyO3/pyo3) 几乎是无缝让 Rust 的代码可以为 Python 所用。感兴趣的话可以看 polars，它是更高效更节省内存的 pandas: https://github.com/pola-rs/polars

1

bearlu
2021-08-25

老师，请教个问题，所有存储在栈中的数据都必须拥有一个已知且固定的大小。这个式rust规则，还是操作系统的规则？

展开

作者回复: 这是编写安全代码的基本规则，rust 将它严格地应用在编译器中。因为不固定大小的内存，比如 varargs，alloca 分配在栈上，会带来很多副作用。你可以看看我之前的回复。

2021-08-25

progarmming rust，rust编程之道得到电子书都有。开始rustling，不错

作者回复: 加油！

qinsi

2021-08-25

Rust里循环引用就会内存泄漏（比如用RefCell<Rc<T>>），但简单的循环引用只要是mark-and-sweap的GC都可以处理。如果应用里有很多带循环引用的数据结构，用Rust写带来的心智负担就比用带GC的语言写大很多。

展开

作者回复: 对于引用计数引发的循环引用，可以用 weak 解决：https://doc.rust-lang.org/std/rc/struct.Weak.html。循环引用的使用场景确实 tracing GC 处理起来更简单。

茶底

2021-08-25

看过一边rust book已然忘却了

展开

作者回复: 可以在学习这个课程的时候再捡起来

王硕尧

2021-08-25

Jon Gjengset 正在写的 Rust for Rustaceans 不错，还有Rust in Motion，Rust in Acti

on

作者回复: 嗯，Rust for Rustaceans 比较适合有一定基础的，我也期待这本书。

2021-08-25

现在一些cncf的项目就是go+rust这种模式。比如tidb 还有linkerd 2。感觉这种相互配合的模式就挺好的。rust未来可期。

展开

作者回复: 嗯，很多项目一开始是纯 golang，后来会把一些核心的，需要高性能的部分抽取出来用 rust 实现。

1